ParaPLL: Fast Parallel Shortest-path Distance Query on Large-scale Weighted Graphs

Kun Qiu
School of Computer Science
Fudan University
Shanghai, China
qkun@fudan.edu.cn

Yuanyang Zhu
School of Computer Science
Fudan University
Shanghai, China
yyangzhu17@fudan.edu.cn

Jing Yuan
School of Computer Science
Fudan University
Shanghai, China
jyuan16@fudan.edu.cn

Jin Zhao
School of Computer Science
Fudan University
Shanghai, China
jzhao@fudan.edu.cn

Xin Wang
School of Computer Science
Fudan University
Shanghai, China
xinw@fudan.edu.cn

Tilman Wolf
Department of Electrical and Computer Engineering
University of Massachusetts Amherst
Amherst, Massachusetts
wolf@umass.edu

ABSTRACT

Determining the shortest-path distance between vertices in the weighted graph is an important problem for a broad range of fields, such as context-aware search and route selection. While many efficient methods for querying shortest-path distance have been proposed, they are poorly suited for parallel architectures, such as multi-core CPUs or computer clusters, due to the strong task dependencies. In this paper, we propose ParaPLL, a new parallelism-friendly framework for fast shortest-path distance query on large-scale weighted graphs. ParaPLL exploits intra-node and inter-node parallelism by using shared memory and message passing paradigms respectively. We also design task assignment and synchronization policies, which allow ParaPLL to reach remarkable speedups compared to state-of-the-art solutions. Moreover, we also prove the correctness of ParaPLL. To the best of our knowledge, ParaPLL is the first parallel framework that utilizing pruned landmark labeling to accelerate shortest-path distance queries on large-scale weighted graphs. Our evaluation results show that ParaPLL is 9.46 times faster than the corresponding serial version on a weighted $0.3M$-vertex graph using a 12-core computer. ParaPLL on a 6-node computer cluster can also achieve a speedup of up to 5.6 over the single-node implementation.

CCS CONCEPTS

• **Computing methodologies → Massively parallel algorithms**;
• **Theory of computation → Shortest paths**;

KEYWORDS

shortest-path distance query, large-scale graphs, MPI

ACM Reference Format:
Kun Qiu, Yuanyang Zhu, Jing Yuan, Jin Zhao, Xin Wang, and Tilman Wolf. 2018. ParaPLL: Fast Parallel Shortest-path Distance Query on Large-scale Weighted Graphs. In *ICPP 2018: 47th International Conference on Parallel Processing, August 13–16, 2018, Eugene, OR, USA.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3225058.3225061

1 INTRODUCTION

Shortest-path distance is one of the most fundamental metrics for determining relationships between vertices in a graph. Many practical characteristics of vertices, such as influence and similarity, are relevant to shortest-path distance. For example, the distance between two users can represent closeness in a social network, which can then be used in a social-aware search to help find related content or users [25]. Distance can also indicate the relevance of two web pages in web graphs [21], which can be used in a context-aware search to recommend related pages [5]. Of course, the result of a distance query can also be used for optimal path selection between two nodes in a network [6].

The original approaches for querying distance were based on a breadth-first search (BFS) algorithm (for unweighted graphs) or Dijkstra's algorithm (for weighted graphs). However, these algorithms may encounter scalability issues for large graphs due to their computational complexity of $O(n^2)$ [10]. For example, the time for querying the distance between just two vertices can be estimated as 125ms for $n = 0.1M$ vertices on a modern X5680@3.3GHz CPU (79.92k MIPS, from most recent benchmark measurements) [3], which is too slow for using as a module for context-aware or social-aware search. Low latency is a key requirement in these search scenarios since these applications need real-time interactions [24, 26]. Therefore, it is critical to reduce the time of distance query.

To reduce query time, two-stage algorithms, which include indexing stage and querying stage, have been proposed. A straightforward two-stage solution is to pre-compute, or "index," the distances of all node pairs and to store them in an index. A distance query can then be answered immediately with a single $O(1)$ lookup in the index. However, fast query comes with high time cost in the indexing stage, say, roughly $O(n^3)$ by Dijkstra or Floyd-Warshall.

For example, indexing a graph with $n = 0.1M$ vertices may take more than 12, 500s, which is not practical for large-scale graphs.

Recently, more attentions have been paid to finding methods that can balance the time consumption between indexing and querying [7, 12, 22–24]. Despite the improved efficiency, these two-stage algorithms are still extremely time-consuming in indexing large graphs. It is reported that with an Intel Xeon X5680@3.33GHz, TEDI [23] requires 2, 226s of indexing time for a 0.6M-vertex unweighted graph and HCL [13] requires 253, 104s of indexing time for a 0.7M-vertex unweighted graph.

In order to accelerate the indexing stage, BBQ [24] has been proposed as an efficient path query algorithm based on optimized TEDI and significantly reduces the indexing time via a bottom-top-bottom process. PLL [4], which is based on distance labeling, also called distance-aware 2-hop cover [9], can index an unweighted 0.3M-vertex graph in only 4s. However, both BBQ and PLL can encounter scalability issues in indexing larger-scale weighted graphs, which is a more general case that has higher time/space needs for indexing. For example, BBQ needs more than $96GB$ memory to index a million-node graph. Although the original PLL does not need a lot of memory, the weighted version still needs 1154.63s to index a $0.2M$-vertex weighted graph in our experiments. The lack of parallel capabilities leaves these algorithms unable to build indexes for weighted graphs efficiently on multi-core CPUs and computer clusters. It is hard to parallelize BBQ or PLL since both algorithms have strong task dependencies. While a parallel version of PLL has been proposed, it cannot be used for weighted graphs [11].

There exist several general graph computing frameworks, such as GraphLab or Pregel [16], which can enable parallel processing of graphs. Unfortunately, they cannot be utilized directly to accelerate the indexing stage since it is hard to make these frameworks compatible with the shortest-path distance algorithms.

We believe that it is critically important to develop a native framework to parallelize the query algorithm such that its real-world performance can be maximized on off-the-shelf parallel hardware platforms, such as multi-core CPUs. Towards this objective, we propose ParaPLL, a fast exact shortest-path distance query framework for large-scale weighted graphs. ParaPLL, which is based on parallelizing PLL, addresses the performance challenges in parallel distance-indexing of the shortest path. In our design, ParaPLL can make full use of the computing potentials offered by multiple levels of parallelism, such as multi-core CPUs (i.e., intra-node level) and computer clusters (i.e., inter-node level) to handle large-scale graphs indexing with a few task assignment and synchronization issues. To the best of our knowledge, ParaPLL is the first parallel framework that utilizes pruned landmark labeling to index weighted large-scale graphs in the order of several to hundreds of seconds.

More specifically, ParaPLL can index an AS IPv4 topology with $0.2M$ vertices and weighted edges in 124.48s using 12 computing threads on a single computer, which is 9.46 times faster than the original PLL ($1154.63s$). For a social network graph with $0.1M$ vertices and weighted edges, ParaPLL is with an indexing time of 8.39s also 5.46 times faster than the original PLL. In addition, ParaPLL can also be deployed in a computer cluster. Our evaluation results show that ParaPLL on a 6-node cluster is $2.05 \sim 5.60$ times faster than the corresponding single-node version. Our algorithm achieves this performance through a parallel strategy that eliminates task dependencies. We show that this parallelization strategy does not change the correctness of the results. To further optimize the efficiency of ParaPLL, we explore task assignment policies and synchronizing strategies to decrease the synchronization overhead on intra-node and inter-node level. Our experimental results show that ParaPLL achieves remarkable speedups with multiple levels of parallelism and that our assignment policies and synchronizing strategy can reduce unnecessary synchronization.

The remainder of the paper is organized as follows. Section 2 discusses the problem background and challenges. Section 3 provides an overview of ParaPLL. Section 4 shows the detailed design of ParaPLL. We present our evaluation in Section 5, and we conclude this paper in Section 6.

2 BACKGROUND AND MOTIVATION

In this section, we briefly introduce 2-hop cover and PLL, followed by related work in the context of graph computation frameworks. We then discuss task dependency and the importance of task assignment policies.

2.1 2-hop cover and PLL

As we have mentioned above, ParaPLL is motivated by PLL (Pruned Landmark Labeling), a framework mainly based on 2-hop cover. Intuitively speaking, PLL separates the shortest-path distance query into two stages, called indexing stage and querying stage. PLL first computes some intermediate results by a pruned search algorithm, and then queries the specific distance between two vertices by utilizing the pre-computed intermediate results. Comparing to traditional algorithms that are directly querying distance, PLL can reduce unnecessary intermediate results that are never used when querying shortest-path distance in the querying stage. The 2-hop cover strategy makes PLL several times faster than naïve BFS, Dijkstra.

Table 1: Terms of Definition

Notation	Description
$G = (V, E)$	A undirected graph G with vertex set V and edge set E
n	The number of vertices in G
m	The number of edges in G
$e_{u,v} \in E$	An edge between vertex u and vertex v in G
$\sigma(e_{u,v})$	The weight of $e_{u,v}$
$P(s,t)$	The shortest path from vertex s to vertex t
$\sigma(P(s,t))$	The distance of $P(s,t)$

We use the notation summarized in Table 1 throughout this paper. We define all vertices, except s and t in $P(s,t)$, as internal vertices. It was proven that any non-trivial shortest-path $P(s,t)$ can be separated into two shortest-paths $P(s,u)$ and $P(u,t)$ with a vertex u, which is an internal vertex in $P(s,t)$ [19]. In other words, we can deduce $P(s,t)$ by finding a vertex u in G that minimizes $\sigma(P(s,u)) + \sigma(P(u,t))$. We use vertex u to conjunct two paths into $P(s,t)$. This method, which utilizes existing shortest-paths to find new shortest-path, is called *2-hop cover* [9].

In the indexing stage, for any specific vertex u, we can find a set of paths $\{P(s,u), \forall s\}$ by utilizing Dijkstra's algorithm with the initial vertex u. This is called indexing for vertex u, and the result $\{P(s,u), \forall s\}$ is called label $L(u)$. However, computing all pairs s, u for $P(s,u)$ will take significant time since the time complexity is $O(n^3)$. Obviously, if we use the *2-hop cover* to query shortest-path distance in the querying stage, not all shortest-paths that need to be computed by Dijkstra's algorithm in indexing stage are needed since they can be simply connected by other shortest paths through *2-hop cover*. Thus, if we have computed a path P' from vertex s to vertex u through Dijkstra's algorithm, we can judge whether the distance of P' is larger than the current shortest-path from s to u that can be computed by the *2-hop cover*. If the distance of P' is larger than the current shortest-path from s to u computed by the *2-hop cover*, it will be pruned because P' will never be the shortest path from s to u (and it also cannot be used as any part of a shortest path). This pruning strategy is called *Pruned landmark labeling (PLL)* as it prunes unnecessary labels during the indexing stage. To answer a distance query, PLL uses the *2-hop cover* to compute the distance in indexed labels $\{L(u), \forall u\}$. Intuitively speaking, PLL indexes labels and uses indexed labels to prune unnecessary labels during the indexing stage. After all necessary labels are indexed, PLL can query distance using these indexed labels in querying stage.

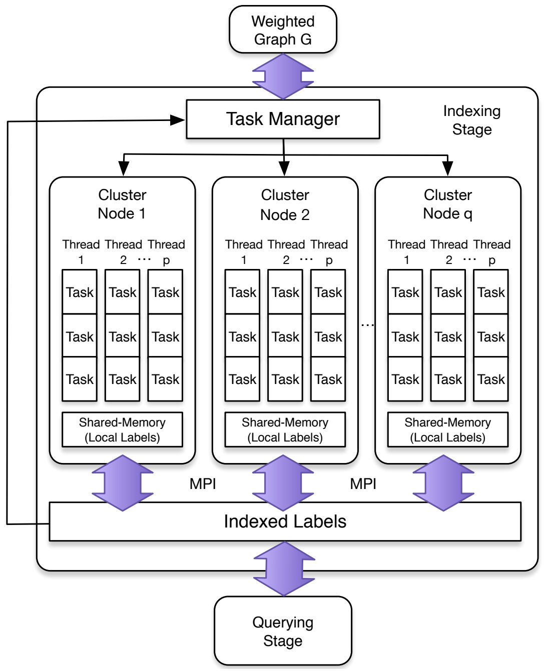


Figure 1: The workflow of ParaPLL.

2.2 Task dependencies and task assignment

PLL uses an iterative algorithm in the indexing stage, which utilizes existing labels to index new labels at each iteration. Although we can split the computing task into several subtasks and assign them to different compute nodes, the original PLL has strong task dependencies. The result of one iteration is highly related to the result of previous iterations, which makes computing tasks in PLL hard to be split. Moreover, the indexing sequence for iterations can also impact the efficiency of the pruning algorithm.

For example, suppose there are two vertices a and b, and indexing vertex a first can help pruning unnecessary labels when indexing vertex b. In contrast, indexing vertex b first does not help to prune any unnecessary labels when indexing vertex a. Thus, indexing vertex a before vertex b can increase the efficiency of the pruned algorithm. Although we can find the optimal indexing sequence in a serial version of PLL, finding the optimal sequence in a parallel version is not an easy task. We can control the indexing sequence only by designing a task assignment policy. Thus, we need to prove the correctness of our task splitting strategy as well as introduce our task assignment policies in following sections.

2.3 Existing graph computation frameworks

In order to increase the performance of the algorithm, we can utilize a computer cluster to increase the performance in the indexing stage. However, it is hard to directly utilize existing graph computation frameworks, such as GraphLab and Pregel [16], that use specific computation models to parallelize PLL, as PLL does not fit these models well. The most important reason is that these frameworks use an out-of-core model, which is also called distributed external memory model. Since PLL is basically a shared-memory model algorithm, all computing processes need to use the current shared existing labels to compute new labels. Thus, directly utilizing out-of-core models still suffers from efficiency hurdles.

As can be expected, utilizing a shared-memory model in a computer cluster will encounter several problems. Regularly synchronizing memory among all compute nodes is still necessary in the cluster or the efficiency of the pruning algorithm will decrease. (If we rarely synchronize indexed labels among all nodes, redundant labels may exist.) Clearly, a tradeoff is also introduced by the communication overhead between different nodes during synchronization. Thus, finding an appropriate synchronization frequency is an important task. Comparing to existing graph computation frameworks, Message Passing Interface (MPI) is a lower-level parallel model that can manually control the memory synchronizing frequency.

Meanwhile, there also exist several distributed query frameworks. DistR [8] is a distributed framework for the reachability query problem in a large-scale uncertain graph. It is efficient to concern communication and computational costs with high accuracy. However, reachability queries are different from shortest-path distance queries, which have smaller use scenarios. Also, gStore [27], which is a graph-based SPARQL query engine, has been proposed. However, it is more like a storage manager for graph data, not aimed at increasing the efficiency of shortest-path distance queries.

3 AN OVERVIEW OF PARAPLL

In this section, we give an overview of the ParaPLL framework.

3.1 Distance query

We use Figure 1 to illustrate the workflow of ParaPLL. As mentioned above, ParaPLL is a two-stage algorithm consisting of an indexing stage (middle part of Figure 1) and a querying stage (bottom part of Figure 1). We first look at the querying stage.

Suppose all indexes (labels) are computed in the indexing stage. We can then answer an incoming distance query with $\textsc{Query}(s,t,L)$, which is defined as follow:

$$min\{\sigma(P(u,s)) + \sigma(P(u,t))|(P(u,s) \in L(s), (P(u,t) \in L(t)\},$$

where $L(s)$ and $L(t)$ are indexed labels. Intuitively speaking, label $L(s)$ and $L(t)$ are shortest paths that ended with vertex s and t. We can find a vertex u that satisfies both $P(u,s)$ in $L(s)$ and $P(u,t)$ in $L(t)$. When joining $P(u,s)$ with $P(u,t)$, we can get a new path from s to t through u. By enumerating all vertices u with s and t in Query, we can obtain the shortest-path distance.

3.2 ParaPLL challenge 1: Computing sequence

The parallel version of this algorithm is very different from the original version. The first challenge is determining a suitable computing sequence. Since the computing sequence can leverage the efficiency of the pruning strategy, finding an optimized sequence is a critical problem. Contrary to the original PLL, for which it is easy to find the optimal computing sequence, finding an optimal sequence for the parallel version is not easy. We introduce a task manager, which can statically or dynamically assign computing tasks to different threads or nodes according to assignment policies. By utilizing this task manager, we can manually control the indexing sequence and find the most appropriate policy to increase the performance of ParaPLL. We describe our assignment policy in Section 4.

3.3 ParaPLL challenge 2: Synchronization

ParaPLL needs to run on a computer cluster with multiple compute nodes (inter-node level parallelism) and on nodes with multiple processor cores (intra-node level parallelism). As mentioned above, the PLL algorithm needs currently indexed labels to prune unnecessary indexed during the computation, making it difficult for implementation on a computer cluster. In ParaPLL, all computing tasks also need current indexed labels to prune unnecessary labels that are newly computed.

Although indexed labels are easily shared at the intra-node level, we need to use MPI to synchronize indexed labels at the inter-node level. We use the task manager to assign tasks to different nodes, which can only utilize a shared memory when we synchronize indexed labels among different nodes using MPI. Thus, finding the most appropriate synchronization interval is another challenge task we need to solve. We discuss our synchronization policy in Section 4.

4 DETAILED DESIGN OF PARAPLL

4.1 The weighted serial version

As mentioned above, we improve the original PLL algorithm to work with weighted graphs. First, we propose the Pruned Dijkstra algorithm shown as Algorithm 1. Different from the original PLL, a priority queue is used to store vertices that are extended by Dijkstra in line 2 of Algorithm 1. Then, we invoke Pruned Dijkstra for each node in V to index all labels.

To estimate the time complexity of the weighted serial version, suppose w denotes the tree-width of G, as the w is a measure of the count of vertices mapped into any tree vertex in an optimal tree decomposition of G. The weighted serial version algorithm takes $O(wm\log^2 n + w^2 n \log^2 n)$ time to index since an enqueue/dequeue operation in a priority queue takes $O(\log n)$.

Algorithm 1: Pruned Dijkstra(G, v_k, L'_k)

```
   Input: graph G, vertex $v_k \in V$, existed labels $L'_k$
   Output: new labels $L'_k$
1  $D$ is an array that save distance from $v_k$
2  $D[v_k] \leftarrow 0$ and $D[v] \leftarrow \infty$ for $v \in V \backslash \{v_k\}$
3  $Q \leftarrow$ a priority queue with one element ~~$u$~~
4  while $Q$ is not empty do
5      Dequeue $u$ from $Q$
6      if $\textsc{Query}(v_k, u) \leq$ ~~$P$~~$[u]$ then
7          continue
8      $L'[u] \leftarrow L'[u] \cup \{(v_k, D[v_k])\}$
9      for $w$ that is the neighbor of $u$ do
10         $D[w] \leftarrow D[u] +$ ~~$1$~~
11         Enqueue $w$ to $Q$
12 Return $L'_k$
```

4.2 The task assignment policy

We need to simultaneously and efficiently perform Pruned Dijkstra from different vertices to index labels in the indexing stage. The key to ensuring the best performance of ParaPLL is to distribute tasks without reducing the efficiency of pruning. In the serial version, the method is to choose the optimal computing sequence for all vertices, which has already been determined in PLL. Similar to the serial version, in ParaPLL we reorder the computing sequence of vertices from higher degree to lower degree since we would like to prune later search as much as possible. The reason is that we can prune a more substantial part of pairs of vertices in Pruned Dijkstra in an earlier stage. That is, labels that are indexed in an earlier stage should provide pruning potential for as many pairs of vertices as possible. We give a briefly explanation in Proposition 2.

To introduce our assignment policies, we start with the basic version, a static assignment method that assigns tasks before the indexing at the intra-node level on only one computer with the shared-memory model. Then, we present a dynamic assignment method, which assigns tasks according to the current labels during the indexing. Finally, we scale our assignment policies to the inter-node level across a computer cluster.

4.3 ParaPLL with static assignment

At the intra-node level, we use a shared-memory model with multithreading. In the static assignment policy, all vertices are assigned to one of multiple threads before indexing. For example, Figure 2 shows 3 threads and 9 vertices, ordered from higher degree to lower degree. Thread 1 executes Pruned Dijkstra with $v1, v4, v7$. At the same time, thread 2 executes Pruned Dijkstra with $v2, v5, v8$, and thread 3 executes Pruned Dijkstra with $v3, v6, v9$.

We define the static assignment policy as assigning vertices in V from higher degree to lower degree to all threads before indexing.

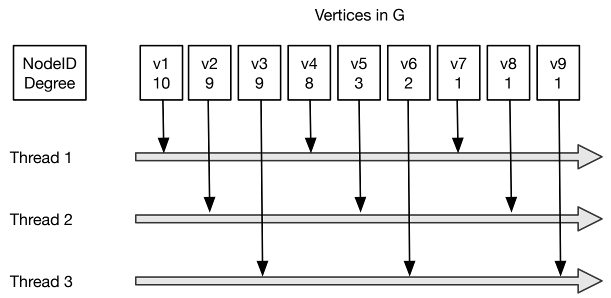


Figure 2: The static assignment policy.

From Figure 2 we can see that the task manager uses a round-robin strategy to assign vertex v_k of Pruned Dijkstra. Thus, we run several instances of Pruned Dijkstra for different vertices simultaneously, each Pruned Dijkstra computing only the vertex that was assigned by the task manager. If the path that is found by the algorithm is larger than the path that can be computed by indexed labels through Query, we can prune the found path. Otherwise, we add the newly found path to the current labels.

We prove the correctness of this static assignment policy:

Proposition 1. Labels that created by ParaPLL can always answer correct shortest-path distance.

We can prove Proposition 1 by mathematical induction.

Proof 1. L_0 is a special label since it is an empty set. Assume that L_k can be obtained by invoking Pruned Dijkstra k times.

(1) **Condition 1**: If only one thread exists, the situation equals to the serial version PLL. Labels for answering correct shortest-path distance can be obtained eventually [4].
(2) **Condition 2**: If there are two threads. Suppose L_k is already obtained, and Pruned Dijkstra is invoked with v_{k+1}, v_{k+2} simultaneously. For v_{k+1}, the situation is same to the serial version PLL, L_k can be used to prune the vertex in Pruned Dijkstra. Thus, the label L_{k+1} can be obtained from v_{k+1}. For v_{k+2}, the computation lacks some labels in L_{k+1} that is utilized for pruning strategy when indexing v_{k+2} since the indexing of v_{k+1} may not be finished. This can lead to some redundant labels when invoking Pruned Dijkstra with v_{k+2}. However, according to the Query, the answer is obtained by choosing the minimum sum of two sub-paths, making redundant labels do not influence the correctness of answering a distance query requirement. Thus, the label L_{k+2} which can answer correct shortest-path distance, can be obtained eventually.
(3) **Condition 3**: Suppose there are more than three threads. (a) If there are exactly three threads, Pruned Dijkstra is invoked with $v_{k+1}, v_{k+2}, v_{k+3}$ simultaneously. According to Condition 2, the label L_{k+1}, L_{k+2} can be obtained. If v_{k+1}, v_{k+2} are regarded as one block, it can also fit Condition 2. Thus, the label L_{k+3} that can answer correct shortest-path distance, can be obtained eventually. (b) If there are p threads ($p > 3$), $v_{k+1}, v_{k+2}, \ldots, v_{k+p-1}$ can be regarded as one block. According to Condition 2, the final label L_{k+p} which can answer correct shortest-path distance can be obtained eventually.

After all threads are finished, all labels that can answer correct shortest-path distance are obtained.

In order to estimate the efficiency loss of the pruning strategy, we use $\psi(v_i)$ to indicate the pruning efficiency that is obtained by indexing v_i [18]. $\psi(v_i)$ is the number of shortest paths that pass through v_i in G. For any v_i, v_j, it is proved that indexing v_i first can achieve higher pruning efficiency if $\psi(v_i) > \psi(v_j)$ [18]. Thus, the computing sequence $v_1, v_2, \ldots, v_n$ with $\psi(v_1) \geq \psi(v_2) \geq \cdots \geq \psi(v_n)$ is an optimal sequence that can obtain maximal pruning efficiency in the serial version PLL [4].

Proposition 2. Suppose there are p threads with n vertices. The maximal efficiency loss is not larger than $\sum_{i=1}^{1+kp}\{\psi(v_{i+p}) - \psi(v_i)\}, k < \left\lfloor \frac{n}{p} \right\rfloor - 1$ by ParaPLL, while $v_1, v_2, \ldots, v_n$ with $\psi(v_1) \geq \psi(v_2) \geq \cdots \geq \psi(v_n)$ is an optimal sequence in serial version PLL.

Proof 2. For a specific i, suppose $v_i, v_{i+1}, \ldots, v_{i+p}$ are currently assigned to p threads by static assignment policy. Since the computing sequence is out-of-order, the worst case of computing sequence is $v_{i+p}, \ldots, v_{i+1}, v_i$. By first indexing v_{i+p} and then v_i, we cause an efficiency loss of $\psi(v_{i+p}) - \psi(v_i)$. Supposing all computing sequences are in worst-case order, the efficiency loss is $\sum_{i=1}^{1+kp}\{\psi(v_{i+p}) - \psi(v_i)\}, k < \left\lfloor \frac{n}{p} \right\rfloor - 1$.

From Proposition 2, we can see that there are two ways to decrease the efficiency loss: (1) reducing the number of threads p or (2) determining an assignment policy that can generate a computing sequence that is closest to the optimal sequence. Clearly, the first option should not be applied since it reduces parallelism and increases the indexing time. Thus, in order to achieve the goal of the second option, we design a dynamic assignment policy.

4.4 ParaPLL with dynamic assignment

In this subsection, we propose a dynamic assignment policy to replace the static assignment policy to obtain better performance. In the static assignment policy, we assign each vertex to a specific thread before indexing, which means that tasks in each thread could not be changed. Thus, the total execution time entirely depends on the slowest thread.

To address this problem, we design a dynamic assignment policy that makes all threads compete for the vertex with the highest degree for computing. Intuitively speaking, the task manager dynamically assigns the vertex with the highest degree in V that does not be indexed to the thread that is free to index. In our observation, the dynamic assignment policy performs very efficiently.

Since we do not have the exact workload of Pruned Dijkstra for each vertex, we need to design a fair strategy to minimize the longest execution time. We design a queue in the task manager, reorder vertices from higher degree to lower degree, and put them into a queue. Then, we propose Algorithm 2. Only one thread can pick up a vertex from the vertices queue at any time. If the indexing task in a thread is finished, the thread can fetch a new task from task manager.

We show an example in Figure 3. In the figure, vertices are assigned to threads dynamically, and we use the length of the arrow to indicate how much time the thread uses to finish its task. First, $v1, v2, v3$ are assigned to threads 1, 2 and 3. Then, $v4$ is assigned to

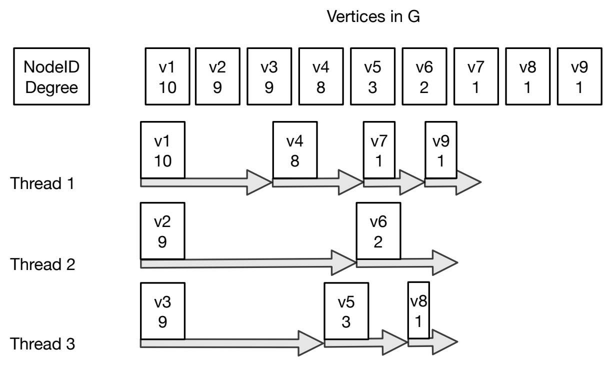


Figure 3: The dynamic assignment policy.

thread 1 as it finished the work for $v1$. As the same reason, $v5$ is assigned to thread 3, $v6$ is assigned to thread 2. At last, we assign $v7$ to thread 1, $v8$ to thread 3 and $v9$ to thread 2.

Algorithm 2: ParaPLL with dynamic assignment policy

```
  Input: Graph $G$, number of thread $p$
  Output: $L$
1 $L'[v] \leftarrow \emptyset$ for $v \in V$
2 $Q$ := a queue with n elements (which are ordered vertices)
3 A semaphore with lock/unlock ensures that only one thread
  can update the label at any time to eliminate race conditions
4 for $k = 1, 2, ..., p$ in parallel do
5     Lock($Q$)
6     Dequeue $v$ from $Q$
7     Unlock($Q$)
8     $L \leftarrow$ PRUNED DIJKSTRA($G, v, L$)
9 Return $L$
```

4.5 Scaling ParaPLL to computer clusters

The static and dynamic assignment policies are designed for intra-node level parallelism. In this subsection, we introduce the inter-node assignment policy of ParaPLL.

Similar to the intra-node level, we assign all tasks to different threads in different compute nodes using the task manager. Within each node, we use the multi-thread model to index vertices with the static or dynamic assignment policy, which is same as the single computer version. However, we need to use MPI to synchronize labels between nodes. Comparing to the shared-memory model, synchronizing labels among compute nodes can introduce significant communication overhead. Thus, we do not synchronize labels immediately for all nodes when new labels are indexed. We only store them in the local node and then synchronize them to other nodes at an appropriate time. As discussed above, delayed synchronization does not affect correctness, but only impacts efficiency.

Thus, there exists a tradeoff between the communication overhead and the efficiency of our pruning strategy. As we illustrate in Figure 4, if we synchronize more frequently, we may get indexed labels with less redundant results, which can increase the efficiency of the pruning strategy. In contrast, if we synchronize less frequently, we may get indexed labels with more redundant results. Therefore, the synchronizing frequency makes a significant difference in the efficiency of ParaPLL. In Section 5, we show results that quantify how the synchronization frequency influence the efficiency of ParaPLL.

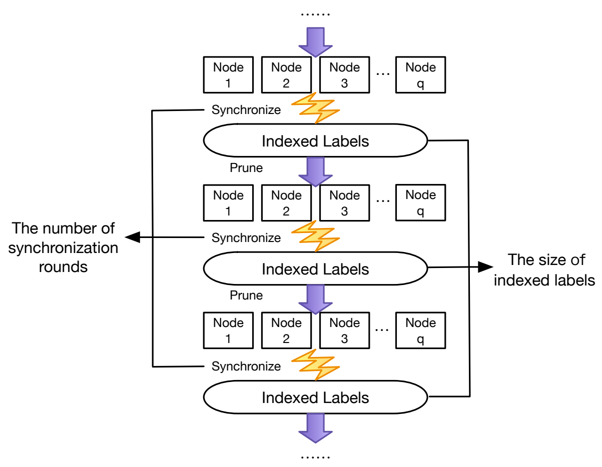


Figure 4: The tradeoff between synchronizing frequency and the size of labels. More synchronizations makes the size of indexed labels less.

We show the cluster version of PRUNED DIJKSTRA in Algorithm 3. Due to the communication overhead, we do not synchronize the label L' to all nodes when the new label are indexed immediately. Instead, we prepare a vector *List* to store all labels indexed during the procedure in line 9. When a new path is computed, we store it into *List* as well as update its local label L. If the specific synchronizing interval that is defined by the task manager has elapsed, we synchronize the labels with the update information in *List* to other computer nodes.

To estimate the time complexity of our algorithm, we assume q nodes in the cluster, each node having p cores, and w denoting the tree-width of G, which is a measure of the count of vertices mapped onto any tree vertex in an optimal tree decomposition of G. We then get a time complexity of $O(wm \log^2 n + w^2 n \log^2 n)/(pq) + O(wn \cdot \log n \log q)$. $O(wm \log^2 n + w^2 n \log^2 n)/(pq)$ is the time for traversing edges and pruning. For the communication overhead, a broadcasting operation consists of $\log q$ times send/receive operation and each cluster node stores indexed labels with $O(w(n/q) \log n)$ space. Thus, $O(wn \cdot \log n \log q)$ is the time for distributing indexed labels.

5 EVALUATION

In this section, we evaluate ParaPLL on commodity hardware. We conduct experiments on a Linux server with dual Intel Xeon E5-2600 (2.5 GHz, 6 cores) with 64 GB DDR3 memory for the single computer evaluation, and we use a computer cluster that consisted of 6 compute nodes with single Intel Xeon E5-2600 (2.5 GHz, 6 cores) and 32 GB DDR3 memory for the cluster evaluation. All algorithms

Algorithm 3: Cluster version Pruned Dijkstra

```
Input: graph G, vertex $v_k$, existed private label $L'_k$
Output: $L'_k$
1  $D[v_k] \leftarrow 0$ and $D[v] \leftarrow \infty$ for $v \in V \backslash \{v_k\}$
2  $List \leftarrow \emptyset$ a set used to store updating information
3  $Q \leftarrow$ a priority queue with one element $u$
4  while $Q$ is not empty do
5      Dequeue $u$ from $Q$
6      if $QUERY(v_k, u) \leq D[u]$ then
7          continue
8      $Lock(L'_k)$
9      $L'_k[u] \leftarrow L'_k[u] \cup \{(v_k, D[v_k])\}$
10     $List \leftarrow List \cup (u, v, D[v_k])$
11     $Unlock(L'_k)$
12     for $w$ that is the neighbor of $u$ do
13         $D[w] \leftarrow D[u] + 1$
14         Enqueue $w$ to $Q$
15 Synchronize $(L'_k)$ with all $List$ gathered from other computers
16 Return $L'_k$
```

Table 2: Dataset

Dataset	n	m	Graph Type
Wiki-Vote	7,115	201,524	Social
Gnutella	10,876	79,988	Internet P2P
CondMat	23,133	186,936	Collaboration
DE-USA	49,109	121,024	Road network
RI-USA	53,658	137,579	Road network
AS-Relation	57,272	983,610	Autonomous Systems
HI-USA	64,892	152,450	Road network
Epinions	75,879	811,480	Social
AskUbuntu	137,517	508,415	Social
Skitter	192,244	1,218,132	Autonomous Systems
Euall	265,214	730,051	Email Communication

are implemented in C++. We use Pthread to implement multi-threading and OpenMPI to implement MPI.

5.1 Dataset

To evaluate the efficiency of ParaPLL, we conduct experiments on a variety of real-world graphs of the following types: social networks, road networks, web networks, and autonomous system (AS) networks. All networks are undirected, weighted graphs. We present the characteristics of these graphs in Table 2. We also show the degree distribution of these real-world graphs in Figure 5. As road networks are typical grid networks, the degree of each vertex in a road network is low in general. Other graphs obey the power law degree distribution. The details for each of these graphs are as follows:

- **Wiki-Vote** [14]: a graph created from the raw Wikipedia administrator election data.

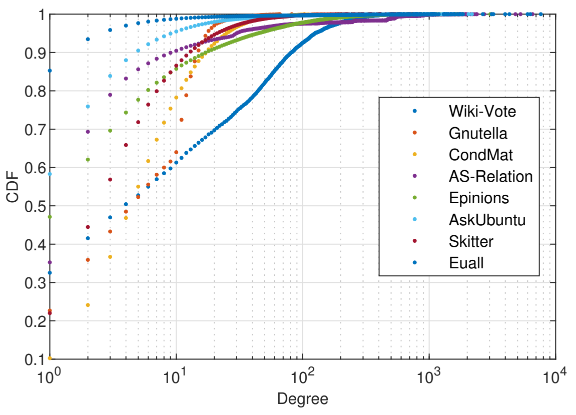


Figure 5: Vertex degree distribution.

- **Gnutella** [15]: a graph created from a snapshot of the Gnutella sharing network collected on Aug. 4, 2002.
- **CondMat** [15]: a graph created from arXiv COND-MAT (Condense Matter Physics) collaboration network, where each vertex is a scientist and each edge represents a co-author relationship.
- **DE-USA** [2]: a road network of Delaware created on May 3, 2002.
- **RI-USA** [2]: a road network of Rhode Island created on May 3, 2002.
- **AS-Relation** [1]: an AS relationships dataset.
- **HI-USA** [2]: a road network of Hawaii created on May 3, 2002.
- **Epinions** [20]: a graph created from the online Epinions social network (www.epinions.com), where each vertex represents a user and each edge represents a trust relationship.
- **AskUbuntu** [17]: a temporal network of interactions on the stack exchange web site Ask Ubuntu (askubuntu.com), where each vertex is a user and each edge represents one interaction between two users.
- **Skitter** [1]: a router-level AS network topology.
- **EuAll** [15]: a network based on email data from a large European research institution.

5.2 Performance at intra-node level

First, we present the performance of our static/dynamic assignment policy on these graphs at the intra-node level (i.e., single computer) to show the efficiency. Table 3 and Table 4 represent the performance of ParaPLL on the graph datasets. IT denotes the indexing time at the indexing stage, SP denotes the speedup, and LN denotes the average label size for each vertex. We also list the performance of serial weighted PLL that we implemented in C++. We do not present memory size since memory size consumed entirely depends on label size and the scale of the graph. ParaPLL does not consume significant extra memory compared to the original PLL – just several arrays of length $|v|$ within each thread. The PLL memory size consumption has a linear relationship with $(n \cdot LN)$ [4]. In our evaluation, the memory usage is not larger than $2.2GB$.

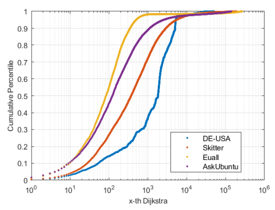
Table 3: Performance of ParaPLL (static assignment policy) compared with PLL for the real-world datasets. IT denotes indexing time, LN denotes average label size, SP denotes speedup compared to single thread.

Dataset	PLL	Number of threads							PLL	Number of threads						
		1	2	4	6	8	10	12		1	2	4	6	8	10	12
	IT(s)	IT(s)	SP	SP	SP	SP	SP	SP	LN	LN	LN	LN	LN	LN	LN	LN
Wiki-Vote	2.21	2.30	0.92	1.25	2.27	2.33	2.51	2.20	70	70	81	82	81	83	81	83
Gnutella	90.66	121.04	1.82	2.96	4.14	5.98	7.66	9.42	302	302	313	319	321	311	315	310
CondMat	23.76	27.04	1.74	2.21	3.37	4.68	5.04	5.39	110	110	114	114	116	117	115	118
DE-USA	191.07	188.84	1.86	3.16	3.20	4.57	5.18	5.88	299	299	311	312	315	317	318	320
RI-USA	367.51	379.76	1.92	2.25	3.51	4.53	5.74	6.69	349	349	357	375	374	378	379	376
AS-Relation	18.13	20.27	1.79	3.38	3.29	3.84	3.96	4.07	60	60	61	62	64	65	65	67
HI-USA	32.10	37.83	1.49	1.70	1.91	2.89	3.19	3.10	145	145	152	158	163	160	160	163
Epinions	159.54	138.58	1.67	2.93	4.06	5.47	6.86	7.41	119	119	121	123	122	124	124	126
AskUbuntu	44.22	45.49	1.86	1.90	3.19	4.13	4.34	4.41	54	54	54	57	56	57	57	59
Skitter	1154.63	1170.29	1.69	2.91	4.80	5.35	7.20	8.31	229	229	233	234	232	237	234	236
Euall	101.09	106.91	1.54	2.31	3.07	3.60	4.81	5.12	66	66	70	69	69	69	69	68

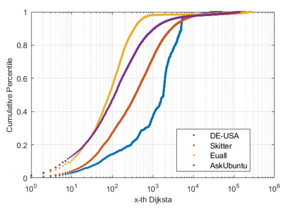
5.2.1 Indexing time. From Table 3 and Table 4, we can see that the indexing time of ParaPLL with a single thread almost equals that of PLL. Then, with an increase in the number of threads, ParaPLL shows a nearly linear speedup on most graphs. Overall, ParaPLL can achieve a speedup of 2.20 $\sim$ 9.42 over a single thread on a 12-core computer. We show that the static assignment policy is 8.31 times faster than the single thread version on Skitter.

Table 4 shows the speedup when we utilize the dynamic assignment policy. With the increase in the number of threads, ParaPLL has a linear speedup, except for Wiki-Vote and HI-USA (only 2.51 to 2.20 in Wiki-Vote, 3.19 to 3.10 in HI-USA). The speedup ratios increase about 1.1 $\sim$ 2.1 times over the static assignment policy. Notably, ParaPLL is 9.46 times faster on Skitter than the original PLL.

5.2.2 Label size. From Table 3, we see that the average label size for each vertex is similar to PLL. Moreover, with an increase in the number of threads, the average label size increases a small amount, which can avoid the time complexity increasing in the querying stage. From Table 4, we can see that the label size decrease by a small amount when using the dynamic assignment policy.



(a) The original PLL



(b) ParaPLL with dynamic assignment

Figure 6: Cumulative distribution of the number of vertices in x-th PRUNED DIJKSTRA

5.3 Performance at inter-node level

We present the performance of ParaPLL at the inter-node level (i.e., computer cluster) on these datasets to show the scalability and efficiency of the algorithm. The task assignment among different nodes is static. We first evaluate ParaPLL with different synchronization frequencies. We increase the number of synchronizations from $c = 1$ to 128, and we invoke a synchronization every $\lfloor \frac{n}{c} \rfloor$ vertices that are indexed by PRUNED DIJKSTRA. Figures 7 (a) and (b) show how synchronization frequency influences the indexing time and the size of labels with 6 compute nodes in the cluster. In Figure 7 (c) (d), we also break down the indexing time by communication time and computation time to highlight the communication overhead.

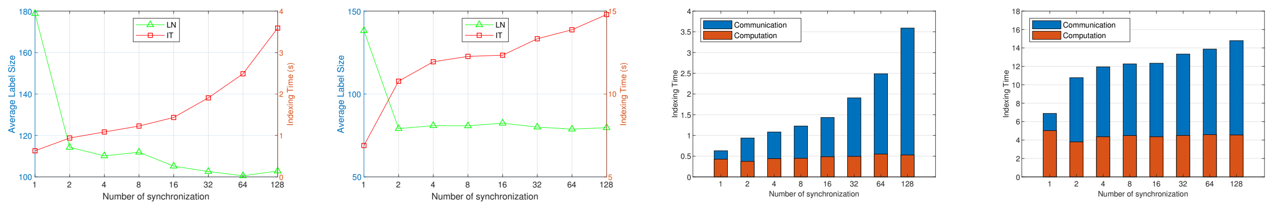
We see that synchronizing indexed labels only once makes ParaPLL most efficient. Thus, we evaluate the performance of ParaPLL with only one synchronization of indexed labels in Table 5. We show the indexing time consumption with both static and dynamic assignment policies for each node.

5.3.1 Indexing time. From Figure 7, we can see that the efficiency decreases significantly when we increase the synchronization frequency. From Table 5, we can see that our framework achieves linear speedups with and increasing number of nodes when we utilize the dynamic assignment policy for each node. Meanwhile, we can find that the performance is better if we utilize the dynamic assignment policy rather than the static policy. Since the scale of Wiki-Vote is small, ParaPLL does not perform well due to the synchronization overhead. However, for the other datasets, we can get a speedup of 2.05 $\sim$ 5.60 with a 6-node cluster compared to a single node using the dynamic assignment policy.

5.3.2 Label size. From Table 5, we can see that the average label size of each vertex increases twofold to threefold. Although the label size increase also increases the query cost by several microseconds, the indexing time still achieves a high speedup in the indexing stage. Moreover, from Figure 7, we see that the indexing size also decreases with an increasing synchronization frequency.

Table 4: Performance of ParaPLL (dynamic assignment policy) compared with PLL for the real-world datasets. IT denotes indexing time, LN denotes average label size, SP denotes speedup compared to single thread.

Dataset	PLL	Number of threads							PLL	Number of threads						
		1	2	4	6	8	10	12		1	2	4	6	8	10	12
	IT(s)	IT(s)	SP	SP	SP	SP	SP	SP	LN	LN	LN	LN	LN	LN	LN	LN
Wiki-Vote	2.21	2.38	1.52	2.88	2.70	3.79	4.59	3.89	70	70	72	73	74	75	77	76
Gnutella	90.66	114.85	1.81	3.87	5.86	7.67	9.27	10.58	302	302	304	304	304	304	304	305
CondMat	23.76	24.78	1.45	2.78	3.99	4.88	5.61	6.43	110	110	112	110	112	112	114	114
DE-USA	191.07	180.66	1.94	3.49	5.10	6.50	7.26	8.02	299	299	300	302	304	305	306	306
RI-USA	367.51	342.84	1.76	3.59	5.25	6.11	7.16	8.03	349	349	351	353	355	357	357	358
AS-Relation	18.13	20.63	1.89	3.15	4.45	5.19	5.50	5.62	60	60	60	62	63	66	64	66
HI-USA	32.10	39.84	1.85	2.91	4.41	4.32	4.41	4.63	145	145	146	148	148	148	149	149
Epinions	159.54	143.56	1.61	3.38	5.04	6.57	7.79	8.93	119	119	120	121	122	122	123	124
AskUbuntu	44.22	45.82	1.94	3.15	4.56	5.02	5.36	5.46	54	54	54	55	55	56	57	58
Skitter	1154.63	1177.57	1.87	3.72	5.41	6.93	8.08	9.46	229	229	229	229	229	229	230	230
Euall	101.09	108.47	1.79	3.47	4.38	5.29	5.79	6.15	66	66	66	66	66	66	66	66



(a) Wiki-Vote (b) Askubuntu (c) Wiki-Vote (d) Askubuntu

Figure 7: Performance of ParaPLL with different synchronization frequency. (a) (b) show how the synchronization frequency influences the label size and indexing time. (c) (d) break down the index time by communication and computation.

5.4 Results summary and analysis

In this subsection, we provide an analysis of the characteristics of ParaPLL from our evaluation results.

5.4.1 *Pruned Dijkstra.*

As we have mentioned, the number of intermediate indexed results can be indicated by the size of labels. Figure 6 shows the cumulative distribution of the number of indexes between the original PLL and ParaPLL. From Figure 6, we can confirm that almost 90% of distances are added into labels after invoking Pruned Dijkstra a hundred times, which is similar to the original PLL. In other words, there is no apparent pruning efficiency gap between ParaPLL and the original PLL.

5.4.2 *Static and dynamic assignment policy.*

From Table 3, Table 4 and Table 5, we can see that ParaPLL with dynamic assignment policy performs better than the static assignment policy. With the static assignment policy, the performance does not follow a linear speedup in some graphs. The reason is that the static assignment policy cannot balance the execution time among threads, while the dynamic assignment policy can balance them to nearly equal the total processing time in all threads, makes the computing sequence more similar to the optimal sequence. In other words, with the dynamic assignment policy, all cores are fully utilized during indexing, which makes ParaPLL perform better.

5.4.3 *Synchronization frequency.*

From Figure 7 (a), we can see that with the increasing frequency of the synchronization, the indexing time becomes larger while the label size becomes smaller. Also, in Figure 7 (b), the label size does not increase. In Figures 7 (c) and (d), we can see that although a small label size may increase the efficiency of the pruning strategy (e.g., when we synchronize twice), we find that the significant communication overhead can cover this advantage. We have to stop all threads to synchronize labels among all compute nodes in the cluster. Then, all nodes in the cluster have to broadcast and receive labels to/from other nodes, which takes $O(lq \log q)$ time complexity for one synchronization (q denotes the number of cluster nodes, l denotes the size of labels to broadcast). Thus, synchronizing among compute nodes costs significant time. This leads to the conclusion that a few synchronizations are the best option, which provides a balance between low label size and low communication overhead.

6 CONCLUSION

In this paper, we have proposed ParaPLL as a solution to address the scalability bottleneck in the shortest-path distance query problem. We have designed a framework that utilizes multiple levels of parallelism (intra-node and inter-node) to accelerate the indexing stage in shortest-path distance query. ParaPLL integrates the following features into its overall design: first, ParaPLL eliminates

Table 5: Performance of ParaPLL for the real-world datasets.
IT denotes indexing time, LN denotes average label size, SP denotes speedup compared to single node.

Dataset	Number of compute nodes (static policy)						Number of compute nodes (dynamic policy)											
	1	2	3	4	5	6	1	2	3	4	5	6	1	2	3	4	5	6
	IT(s)	SP	SP	SP	SP	SP	IT(s)	SP	SP	SP	SP	SP	LN	LN	LN	LN	LN	LN
Wiki-Vote	0.96	1.25	1.43	1.58	1.80	1.97	0.92	1.43	1.49	1.71	1.79	2.05	71	99	125	146	162	178
Gnutella	19.29	1.76	2.82	3.41	4.24	5.55	17.87	1.86	3.08	3.76	4.71	5.60	302	436	527	598	656	707
CondMat	8.02	1.63	1.77	1.83	2.13	2.86	7.21	1.57	2.13	2.31	2.63	2.93	111	166	208	244	273	299
DE-USA	61.66	1.85	1.45	2.07	1.92	2.71	46.95	1.72	1.25	1.92	1.69	2.35	303	436	591	647	747	786
RI-USA	100.15	1.69	2.55	2.24	3.15	3.72	88.10	1.64	2.67	2.86	4.32	4.91	354	503	604	695	758	820
AS-Relation	7.41	1.52	2.14	2.27	2.64	2.82	6.79	1.76	2.20	2.44	2.84	3.05	62	80	96	108	115	126
HI-USA	22.34	1.75	2.36	2.42	2.85	3.07	18.34	1.54	2.23	2.12	2.87	3.21	147	197	226	258	274	287
Epinions	29.80	1.63	2.28	2.68	3.16	3.52	29.44	1.84	2.54	3.02	3.65	4.04	121	183	228	271	303	336
AskUbuntu	15.48	1.50	1.75	1.91	2.01	2.47	14.83	1.56	1.74	2.22	2.46	2.61	55	80	99	114	126	138
Skitter	228.89	1.89	2.73	3.50	3.77	5.06	219.72	1.95	2.83	3.73	4.06	5.10	229	329	409	472	533	585
Euall	35.19	1.72	2.45	3.04	3.27	3.95	33.93	1.80	2.26	3.02	3.53	3.95	66	85	97	106	112	116

task dependencies while maintaining provable correctness for PLL; second, ParaPLL uses a task manager with two assignment policies to further improve its performance; third, ParaPLL finds an appropriate synchronization frequency for reducing communication overhead. Our evaluation results show that ParaPLL can achieve significant speedups over the original PLL, both on a single computer with multi-core CPUs and on a computer cluster. Comparing ParaPLL with state-of-the-art solutions, the results demonstrate the effectiveness of our framework.

ACKNOWLEDGMENTS

The work was partially supported by Natural Science Foundation of China (61571136), 863 program (2015AA016106) and the U.S. National Science Foundation (1421448). Jin Zhao is the corresponding author.

REFERENCES

[1] 2018. CAIDA. (2018). http://www.caida.org/data

[2] 2018. USA Road Network. (2018). http://www.diag.uniroma1.it/challenge9/data/tiger/

[3] 2018. Xeon X5680 Benchmark. (2018). http://ranker.sisoftware.net/

[4] Takuya Akiba, Yoichi Iwata, and Yuichi Yoshida. 2013. Fast exact shortest-path distance queries on large networks by pruned landmark labeling. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*. ACM, 349–360.

[5] Lars Backstrom, Dan Huttenlocher, Jon Kleinberg, and Xiangyang Lan. 2006. Group formation in large social networks: membership, growth, and evolution. In *Proceedings of the 12th ACM SIGKDD international conference on Knowledge discovery and data mining*. ACM, 44–54.

[6] Randeep Bhatia, Fang Hao, Murali Kodialam, and TV Lakshman. 2015. Optimized network traffic engineering using segment routing. In *Computer Communications (INFOCOM), 2015 IEEE Conference on*. IEEE, 657–665.

[7] Venkatesan T Chakaravarthy, Fabio Checconi, Prakash Murali, Fabrizio Petrini, and Yogish Sabharwal. 2017. Scalable single source shortest path algorithms for massively parallel systems. *IEEE Transactions on Parallel and Distributed Systems* 28, 7 (2017), 2031–2045.

[8] Yurong Cheng, Ye Yuan, Lei Chen, Guoren Wang, Christophe Giraud-Carrier, and Yongjiao Sun. 2016. Distr: a distributed method for the reachability query over large uncertain graphs. *IEEE Transactions on Parallel and Distributed Systems* 27, 11 (2016), 3172–3185.

[9] Edith Cohen, Eran Halperin, Haim Kaplan, and Uri Zwick. 2003. Reachability and distance queries via 2-hop labels. *SIAM J. Comput.* 32, 5 (2003), 1338–1355.

[10] Thomas H Cormen. 2009. *Introduction to algorithms*. MIT press.

[11] Damir Ferizovic. 2015. *Parallel Pruned Landmark Labeling for Shortest Path Queries on Unit-Weight Networks*. Ph.D. Dissertation. National Research Center.

[12] Ada Wai-Chee Fu, Huanhuan Wu, James Cheng, and Raymond Chi-Wing Wong. 2013. Is-label: an independent-set based labeling scheme for point-to-point distance querying. *Proceedings of the VLDB Endowment* 6, 6 (2013), 457–468.

[13] Ruoming Jin, Ning Ruan, Yang Xiang, and Victor Lee. 2012. A highway-centric labeling approach for answering distance queries on large sparse graphs. In *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. ACM, 445–456.

[14] Jure Leskovec, Daniel Huttenlocher, and Jon Kleinberg. 2010. Predicting positive and negative links in online social networks. In *Proceedings of the 19th international conference on World wide web*. ACM, 641–650.

[15] Jure Leskovec, Jon Kleinberg, and Christos Faloutsos. 2007. Graph evolution: Densification and shrinking diameters. *ACM Transactions on Knowledge Discovery from Data (TKDD)* 1, 1 (2007), 2.

[16] Grzegorz Malewicz, Matthew H Austern, Aart JC Bik, James C Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: a system for large-scale graph processing. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 135–146.

[17] Ashwin Paranjape, Austin R Benson, and Jure Leskovec. 2017. Motifs in temporal networks. In *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining*. ACM, 601–610.

[18] Michalis Potamias, Francesco Bonchi, Carlos Castillo, and Aristides Gionis. 2009. Fast shortest path distance estimation in large networks. In *Proceedings of the 18th ACM conference on Information and knowledge management*. ACM, 867–876.

[19] Kun Qiu, Siyuan Huang, Qiongwen Xu, Jin Zhao, Xin Wang, and Stefano Secci. 2017. ParaCon: A Parallel Control Plane for Scaling Up Path Computation in SDN. *IEEE Transactions on Network and Service Management* 14, 4 (2017), 978–990.

[20] Matthew Richardson, Rakesh Agrawal, and Pedro Domingos. 2003. Trust management for the semantic web. In *International semantic Web conference*. Springer, 351–368.

[21] Antti Ukkonen, Carlos Castillo, Debora Donato, and Aristides Gionis. 2008. Searching the wikipedia with contextual information. In *Proceedings of the 17th ACM conference on Information and knowledge management*. ACM, 1351–1352.

[22] Kai Wang, Guoqing (Harry) Xu, Zhendong Su, and Yu David Liu. 2015. GraphQ: Graph Query Processing with Abstraction Refinement-Scalable and Programmable Analytics over Very Large Graphs on a Single PC.. In *USENIX Annual Technical Conference*. 387–401.

[23] Fang Wei. 2010. TEDI: efficient shortest path query answering on graphs. In *Proceedings of the 2010 ACM SIGMOD International Conference on Management of data*. ACM, 99–110.

[24] Qiongwen Xu, Xu Zhang, Jin Zhao, Xin Wang, and Tilman Wolf. 2016. Fast shortest-path queries on large-scale graphs. In *Network Protocols (ICNP), 2016 IEEE 24th International Conference on*. IEEE, 1–10.

[25] Sihem Amer Yahia, Michael Benedikt, Laks VS Lakshmanan, and Julia Stoyanovich. 2008. Efficient network aware search in collaborative tagging sites. *Proceedings of the VLDB Endowment* 1, 1 (2008), 710–721.

[26] Xiaofei Zhang and Lei Chen. 2017. Distance-aware selective online query processing over large distributed graphs. *Data Science and Engineering* 2, 1 (2017), 2–21.

[27] Lei Zou, M Tamer Özsu, Lei Chen, Xuchuan Shen, Ruizhe Huang, and Dongyan Zhao. 2014. gStore: a graph-based SPARQL query engine. *The VLDB journal* 23, 4 (2014), 565–590.